A General Characterization of Indulgence

R. Guerraoui[1,2] N. Lynch[2]

(1) School of Computer and Communication Sciences, EPFL
(2) Computer Science and Artificial Intelligence Laboratory, MIT

Abstract. An indulgent algorithm is a distributed algorithm that, besides tolerating process failures, also tolerates arbitrarily long periods of instability, with an unbounded number of timing and scheduling failures. In particular, no process can take any irrevocable action based on the operational status, correct or failed, of other processes. This paper presents an intuitive and general characterization of indulgence. The characterization can be viewed as a simple application of Murphy's law to partial runs of a distributed algorithm, in a computing model that encompasses various communication and resilience schemes. We use our characterization to establish several results about the inherent power and limitations of indulgent algorithms.

1 Introduction

Indulgence

The idea of *indulgence* is motivated by the difficulty for any process in a distributed system to accurately figure out, at any point of its computation, any information about which, and in what order, processes will take steps after that point. For instance, a process can usually not know if other processes have failed and stopped operating or are simply slow to signal their activity and will indeed perform further computational steps. More generally, a process can hardly exclude any future interleaving of the processes.

This uncertainty is at the heart of many impossibilities and lower bounds in distributed computing, e.g., [9], and it has been expressed in various forms and assuming specific computation models,e.g., [7, 4, 19]. The goal of this work is to capture this uncertainty in an abstract and general way, independently of specific distributed computing and communication models, be they time-based, round-based, message passing or shared memory.

In short, an *indulgent* algorithm is an algorithm that tolerates this uncertainly. In a sense, the algorithm is *indulgent* towards its environment, i.e., the operating system and the network. These can thus be unstable and congested for an arbitrarily long period of time, during which an unbounded number of timing and scheduling failures can occur.

An obvious class of indulgent algorithms are asynchronous ones [9]. These do not make any assumption on communication delays and relative process speeds. As a consequence, no process can for instance ever distinguish a failed process

from a slow one. However, indulgent algorithms do not need to be asynchronous. In particular, an algorithm that eventually becomes synchronous, after an unknown period of time [7], is also indulgent. Similarly, algorithms that rely on an eventual leader election abstraction, such as Paxos [19], or an eventually accurate failure detector, such as the rotating coordinator algorithm of [4], are also indulgent. Other examples of indulgent algorithms include those that assume a time after which processes execute steps in a certain order [21], or an eventual bound on the ratio between the delay of the fastest and the slowest messages [8], as well as algorithms that tolerate an unbounded number of timing failures [23]. All these non-asynchronous indulgent algorithms have the nice flavor that the assumptions they make about the interleaving of processes can only be used for *liveness*. *Safety* is preserved even if these assumptions do not hold.

All these algorithms are devised in specific models that refer directly to specific failure detector machinery or specific synchrony assumptions, typically assuming a message passing model [10, 6, 24, 13, 22].

Murphy's law

The goal of this work is to characterize the notion of indulgence in a general manner, encompassing various distributed computing models, be they round-based or time-based, as well as various communication schemes, be they shared memory or message passing. By doing so, the goal is to determine the inherent power and limitation of indulgent algorithms, independently of specific models.

To seek for a general characterization of indulgence, it is tempting to consider an abstract approach that looks at *runs* of an algorithm as sequences of *events* that occur at the *interface* between the processes executing the algorithm and the *services* [1] used in the algorithm; each event representing a *step* of a process consisting of a process id, a service id, together with the operation invoked by the process on the service with its input and output parameters.

This is in contrast to an approach where we would look into the internals of the individual services involved in the computation and the automata executed on the processes. While appealing for its generality, the abstract approach is not straightforward as we explain in the paper. In particular, it is not immediate how to devise an abstract characterization without precluding algorithms that assume a threshold of correct (non-faulty) processes. This would be unfortunate for indulgent algorithms typically assume for instance a majority of correct processes [7, 19, 4].

The main contribution of this paper is to characterize indulgence by applying *Murphy's law* to partial runs of an indulgent algorithm. Basically, we characterize indulgence through the following property: if the interleaving I (sequence of process ids) of a partial run R (sequence of steps) of an algorithm A *could* be extended with steps of certain processes and not others, while still be tolerated by the algorithm, then the partial run R can *itself* be extended in A with such steps.

[1] Shared memory object, broadcast primitive, message passing channel, failure detector, clock, etc.

More specifically, we say that an algorithm A is indulgent if, given any partial run R of A and the corresponding interleaving I of processes, if A tolerates an extension I' of I where some subset of processes stop taking steps (resp. take steps) after I, then A does also have an extension of R with interleaving I'. In a sense, partial run R does not provide the processes with enough information to predict the extension of the interleaving I: if some extension of I is tolerated by the algorithm, then this extension can also be associated with an extension of R.

Power and limitation of indulgence

We first show in the paper that our characterization of indulgence is *robust* in the sense that it does not depend on the number of failures tolerated by an algorithm. In short, if an algorithm A that tolerates k failures is indulgent, then the restriction of A to runs with $k-1$ failures is also indulgent.

We then highlight the *safety* aspect of indulgent algorithms. Basically, even if an indulgent algorithm relies on some information about the interleaving of processes to solve some problem, the algorithm can only rely on this information to ensure the *liveness* part of the problem. *Safety* is preserved even if the information is never accurate.

We then proceed to show that an indulgent algorithm A is inherently *uniform*: if A ensures the *correct-restriction* of a safety property P, then A ensures the actual property P. A corollary of this, for instance, is that an indulgent algorithm cannot solve the correct-restriction of consensus, also called *non-uniform* consensus (where a process can decide a different value from a value decided by a failed process) without solving consensus (where no two processes should ever decide different value - uniform agreement). This is not the case with non-indulgent algorithms.

We use our uniformity property to show that certain problems are impossible with indulgent algorithms. In particular, we show that no indulgent algorithm can solve a *failure sensitive* problem, even if only one process can fail and it can do so only initially. In short, a failure sensitive problem is one the specification of which depends on the fact that certain processes takes steps or not after a decision is taken. Failure sensitive problems include some classical ones like *non-blocking atomic commit*, *terminating reliable broadcast*, (also known as the *Byzantine Generals* problem) as well as *interactive consistency*. There are known algorithms that solve these problems but these are not indulgent.

Our reduction from uniformity to the impossibility of solving failure sensitive problems is, we believe, interesting in its own right. By showing that our impossibility applies only to initial failures, and holds even if the algorithm uses powerful underlying services like consensus itself, we emphasize the fact that this impossibility is fundamentally different from the classical impossibility of consensus in an asynchronous system if a process can fail during the computation [9].

Finally, we prove that, given n the number of processes in the system and assuming $n-\lfloor n/x \rfloor$ processes can fail ($x \leq n$) , no indulgent algorithm can ensure a

$x-$*divergent* property using only *timeless* services. In short, a $x-$divergent property is one that can hold for partial runs involving disjoint subset of processes but not in the composition of these runs, whereas a timeless service is one that does not provide any real-time guarantee. We capture here, in a general way, the traditional partitioning argument that is frequently used in distributed computing. Corollaries of our result include the impossibility for an indulgent algorithm using message passing or sequentially consistent objects [18] to (a) implement a safe register [18] if half of the processes can fail, as well as (b) implement k-set agreement if $n - \lfloor n/k \rfloor$ processes can fail.

To conclude the paper, we discuss how, using our notion of indulgence, we indirectly derive the first precise definition of the notion of *unreliable* failure detection [4]. Whereas this notion is now folklore in the distributed computing literature, it has never been precisely defined in a general model of distributed computation.

2 Model

Processes and services

We consider a set Π of processes each representing a Turing machine. The total number of processes is denoted by n and we assume at least 2 processes in the system, i.e., $n > 1$. Every process has a unique identity. Processes communicate through shared abstractions, called *distributed services* or simply *services*. These might include sequentially consistent or atomic objects [18, 15], as well as message passing channels and broadcast primitives [14]. The processes can also consult oracles such as failure detectors [4] about the operational status of other processes, or random devices that provide them with arbitrary values from a random set. Each service exports a set of operations through which it is accessed. For instance [20]:

- A message passing channel exports a *send* and a *receive* operations. The *send* takes an input parameter, i.e., a message, and returns simply an *ok* indication that the message was sent. On the other hand, a *receive* does not take any input parameter and returns a message, possibly *nil* (empty message) if there is no message to be received. Message passing channels differ according to the guarantees on message delivery. Some might ensure that a message that is sent is eventually received by every correct process (the notion of *correct* is recalled more precisely below). Others ensure simply that the message is received if both the sender and the receiver are correct.

- An atomic queue exports a *enqueue* and a *dequeue* operations. The *enqueue* takes an input parameter (an element to enqueue) and returns an *ok* indication. On the other hand, a *dequeue* does not take any input parameter and returns an element in the queue (the oldest), if there is any, or simply *nil* if there is no element in the queue.

– A failure detector exports one *query* operation that does not take any input parameter and returns a set of processes that are suspected to have failed and stopped their execution. In a sense, a failure detector provides information about the future interleaving of the processes. More generally, one could also imagine oracles that inform a process that certain processes will be scheduled before others.

Steps and schedules

Each process is associated with a set of possible states, some of which are initial states. A set of n states, each associated with one process of the system, is called a *configuration*. A configuration composed of initial states is called an *initial configuration*. A process is also associated with an automata that regulates the execution of the process according to a given algorithm.

The system starts from an initial configuration, among a set of possible initial configurations, and evolves to new configurations by having processes execute *steps*. A *step* is an atomic unit of computation that takes the system from a configuration to a new configuration.

Every step is associated with exactly one process. In every step, the associated process accesses exactly one shared service by invoking one of the operations of the service and getting back the operation's reply. (We do not assume here any determinism.) Based on this reply, the process modifies its local state before moving to the next step. [2] The automaton of the process determines, given a state of a process and a reply from the invocation of an operation, the new state of the process and the operation to invoke in the next step of the process.

The visible part of a step, at the interface between a process and a service, is sometimes called an *event*. It is modeled by a process *id*, a service *id*, the *id* of an operation, as well as input and output parameters of the operation's invocation. By language abuse, we also call this a step when there is no ambiguity between the event and the corresponding step.

An infinite sequence of steps S is called a *schedule* and the corresponding sequence of process ids is called the *interleaving* of the schedule S and is denoted by $I(S)$. If the sequence is finite, we talk about a *partial* schedule and a partial interleaving. Sometimes we even simply say a schedule and an interleaving if there is no ambiguity. If a process p has its id in an interleaving I then we say that p *appears* in I.

We say that a (partial) schedule S_2 (resp. an interleaving I_2) is an extension of a partial schedule S_1 (resp. partial interleaving I_1) if S_1 (resp. I_1) is a prefix of S_2 (resp. I_2). We write $S_2 \in E(S_1)$ (resp. $I_2 = E(I_1)$).

[2] Executing a local computation, with no access to a shared service in a given step, is simply modeled by an access to an immutable service.

Runs and algorithms

A *run* (resp. a partial) R is a pair (S, C) composed of a schedule (resp. a partial schedule) S and a configuration C, called the initial configuration of the run R. The interleaving of the schedule S, $I(S)$, is also called the interleaving of the run R, and is also denoted by $I(R)$. We say that a (partial) run $R_2 = (S_2, C)$ is an extension of a partial run $R_1 = (S_1, C)$ (we write $R_2 \in E(R_1)$) if S_2 is an extension of S_1. In this case, $I(R_2)$ is also an extension of $I(R_1)$. We denote by $R/p = (S/p, C)$ the restriction of $R = (S, C)$ to the steps involving only process p.

A process p is *correct* in a run R if p appears infinitely often in the interleaving $I(R)$ of that run R, i.e., p performs an infinite number of steps in R. A process p is said to be *faulty* in a run R if p is not correct in R. We say that a process p *initially fails* in a run R if p does not appear in $I(R)$. We denote the set of faulty processes in a run R (resp. interleaving I) by $\mathit{faulty}(R)$ (resp. $\mathit{faulty}(I)$), and the set of processes that do not take any step in R by $\mathit{faulty}^\star(R)$ (resp. $\mathit{faulty}^\star(I)$).

We model an *algorithm* as a set of runs. If R_i is a partial run of a run $R \in A$, we write $R_i \in^\star A$. The interleavings of the runs of an algorithm A are said to be *tolerated* by A and the set of these interleavings is denoted by $I(A)$.[3]

For instance, in *wait-free* computing [15], an algorithm tolerates all possible interleavings: it has at least one run for every possible interleaving.

It is also common to study algorithms that tolerate a threshold of failures, as we precisely define below.

- We say that A is a *k-resilient* algorithm if $I \in A$ if and only if $\mathit{faulty}(I) < n - k$. That is, $I(A)$ contains exactly all interleavings where at least $n - k$ processes appear infinitely often.

- We say that A is a *$k^\star$-resilient* algorithm if $I \in A$ if and only if $\mathit{faulty}^\star(I) = \mathit{faulty}(I) < n - k$. Every process that appears once in any interleaving I of A appears infinitely often in I. (We capture here the assumption of initial failures.)

We assume that the algorithms are *well behaved* in the following senses. (1) Every partial interleaving tolerated by an algorithm A has a *failure-free* extension also tolerated by A. (2) Let A be any algorithm and $R = (C, S)$ any run of A. If C' is an initial configuration similar to C, except for the initial states of the processes in $\mathit{faulty}^\star(R)$, then $R' = (C', S)$ is also a run of A.

[3] This conveys the idea that the interleaving is chosen by the operating system and not by the algorithm. In some sense, the operating system acts as an *adversary* that the algorithm needs to face and it is common to talk about the interleaving of the adversary.

3 Indulgence

Overview

Informally, an algorithm is *indulgent* if no process, and any point of its computation, can make any accurate prediction about the future interleaving of the processes. For instance, no process can ever declare another process as being faulty or correct.

As we discuss below, it is not trivial to capture this intuition without precluding algorithms that tolerate certain interleavings and not others. Example of these algorithms are t-(or $t^\star$-) resilient algorithms. In such algorithms, certain interleavings are known to be impossible in advance, i.e., before the processes start any computation. As we will explain, a naive definition of indulgence would preclude such algorithms.

- Consider a first glance approach (*characterization 1*) that would declare an algorithm A indulgent if, for any partial run R of A, for any process q, A has an extension of R with an infinite number of steps by q. This clearly captures the idea that no process can, at any point of its computation (say after any partial run R) declare that some other process q is faulty, for q could still take an infinite number of steps (after R) and thus be correct. Although intuitive, this characterization is fundamentally flawed, as we discuss below.

- With characterization 1, we might consider as indulgent an algorithm that relies on the ability of a process to accurately learn that at least *one out of two* processes have failed, or learn that certain processes will perform steps in a round-robin manner, provided they perform future steps. Indeed, characterization 1 above simply says that *any* process q can still take steps in *some* extension of the partial run R. For some pair of processes q_1 and q_2, there might be no extension of R with both q_1 and q_2 taking an infinite number of steps in any arbitrary order.
  In particular, we would like indulgence to express the very fact that any *subset* of processes can still take steps after any point of the computation, i.e., after any partial run R, and in any possible order. In fact, there is an easy fix to characterization 1 that deals with this issue. It is enough to require (*characterization 2*) that, for any partial run R of A, for any subset of processes Π_i, A has an extension of R with an infinite number of steps by all processes of Π_i, in every order. As we discuss below however, this characterization raises other issues.

- Characterization 2 might unfortunately lead us to consider as indulgent an algorithm that relies on the ability for the processes to learn that some specific process *will* take steps in the future. A naive way to prevent this possibility is to also require (*characterization 3*) that, for any partial run R of an indulgent algorithm A, for any subset of processes Π_i, A has an extension of R where no process in Π_i takes any step after R. Characterization 3

however excludes algorithms that assume a threshold of correct processes. As we pointed out earlier, many indulgent algorithms [3, 7, 19] assume a correct threshold: in particular, they assume that every partial run has an extension where a majority of processes take an infinite number of steps.

Characterization

Very intuitively, we cope with the issues above by proposing a definition of indulgence inspired by Murphy's law, which we apply to partial runs. Basically, we declare an algorithm A indulgent if, whenever the interleaving $I(R)$ of any partial run R of A could be extended with a certain interleaving, then R also would. In other words, if the interleaving $I(R)$ of a partial run R has an extension $I' \in I(A)$, then A also has an extension R' of R with the interleaving $I(R') = I'$.

Definition (indulgence). An algorithm A is indulgent if, $\forall I_1, I_2 \in I(A)$ s.t. $I_2 \in E(I_1)$, $\forall R_1 \in A$ s.t. $I(R_1) = I_1$, $\exists R_2 \in A$ s.t. $I(R_2) = I_1$ and $I_2 \in E(I_1)$.

In other words, for any pair of interleavings I_1 and I_2 tolerated by A such that I_2 extends I_1, any partial run R_1 of A, such that $I(R_1) = I_1$, has an extension R_2 in A such that $I(R_2) = I_2$.

Basically, our definition says that no partial run R_1 can preclude any extension R_2 with interleaving I_2, provided I_2 is tolerated by A. The definition does not preclude t-resilient algorithms from being indulgent. This would not have been the case for instance with a definition that would only consider as indulgent an algorithm A such that, for any partial run R of A, for any subset of processes $\Pi_i \subset \Pi$, A has an extension R_1 of R where all processes of Π_i are correct, and an extension R_2 of R where no process in Π_i takes any step after R.

Examples

Clearly, an algorithm that makes use of a perfect failure detector [4] is not indulgent. If a process is detected to have failed in some partial run R, then R cannot be extended with an interleaving including steps of p. In fact, even an algorithm relying on an anonymously perfect failure detector is not indulgent [12]. Such a failure detector might signal that *some* process has failed, without indicating which one. When it does so in some partial run R, this indicates that it is impossible to extend R with a run where all processes are correct. Similarly, an algorithm that uses an oracle which declares some process correct, say from the start [11], would not be indulgent if the algorithm tolerates at least one interleaving where that process crashes.

An obvious class of indulgent algorithms are t-resilient asynchronous ones [9]. Such algorithms do not have any partial run providing meaningful information about the future interleaving of the processes. However, and as we explained in the introduction, indulgent algorithms do not need to be asynchronous. Algorithms that rely (only) on eventual properties (i.e., that hold only after an

unknown periods of time) about the interleavings of the processes are indulgent. These include eventually synchronous algorithms [7], eventual leader-based algorithms [19], rotating coordinator-based algorithms [4], as well as algorithms that tolerate an unbounded number of timing failures [23], or assume eventual interleaving properties [21], or an eventual bound on the ratio between the delay of the fastest and the slowest communication [8].

In the following, we prove three properties of indulgent algorithms: *robustness*, *safety*, and *uniformity*. Later, we will also prove some inherent limitations of indulgent algorithms.

4 Robustness

In short, the robustness aspect (of our definition) of indulgence conveys the fact that if an algorithm A that tolerates t failures is indulgent, then the restriction of A to runs with $t-1$ failures is also indulgent. Before stating and proving this property, we define notions of extensions of an algorithm.

Let A and A' be any two algorithms.

- A' is an *extension* of A if $A \subset A'$. (Every run of A is a run of A'.) We also say in this case that A is a *restriction* of A'.

- A' is a *strict extension* of A if (a) $A \subset A'$ and (b) $\forall R \in A'$ s.t. $I(R) \in I(A)$, $R \in A$. (Every run of A' with an interleaving tolerated by A is also a run of A.) We also say in this case that A is a *strict restriction* of A'.

Proposition 1. *Every strict restriction of an indulgent algorithm is also indulgent.*

Proof. (Sketch) Consider an algorithm A that is a *strict restriction* of A'. We proceed by contradiction and assume that A' is indulgent whereas A is not.

The fact that A is not indulgent means that (a) there are two interleavings I_1 and $I_2 \in I(A)$ such that $I_2 \in E(I_1)$, (b) a partial run $R \in A$ such that $I(R) = I_1$, and (c) (*) A has no extension of R, R', such that $I(R') = I_2$.

The fact that I_1 and $I_2 \in i(A)$ means that there are two runs R_1 and $R_2 \in A$ such that $I(R_1) = I_1$ and $I(R_2) = I_2$.

Since A' is an extension of A, and R, R_1 and R_2 are (partial) runs of A, then R, R_1 and R_2 are also partial runs of A'.

Since A' is indulgent, then A has an extension R' of R such that $I(R') = I_2$.

Finally, since A' is a strict extension of A, then $R' \in A$: a contradiction with (*).

Consider an algorithm A that is t-resilient. Remember than this means that A tolerates all interleavings where at least $n-t$ processes are correct, i.e., $n-t$ processes take an infinite number of steps. The subset of all runs of A where at least $n-t-1$ processes take an infinite number of steps is a $t-1$-resilient

algorithm A' that is a strict restriction of A. The proposition above says that if A is indulgent then so is A'. The same reasoning applies to $t^\star$-resilient algorithms. (Note that robustness does not hold for the naive characterization 3 of indulgence discussed earlier in Section 3.)

5 Safety

The safety aspect of indulgence means, roughly speaking, that, even if an indulgent algorithm relies on some information about the interleaving of processes to solve some problem, the algorithm can only rely on this information to ensure the *liveness* part of the problem, and not its *safety*. We first recall these notions.

Safety and liveness

The specifications of distributed computing problems are typically expressed in terms of predicates over runs, also called properties of runs. An algorithm solves a problem if those predicates hold over all runs of the algorithm.

Informally, a safety property states that *nothing bad should happen*, whereas a liveness property states that *something good should eventually happen* [17, 1] .

Consider a predicate P over runs and a specific run R. We say that P *holds in* R if $P(R) = true$; P does not hold in R if $P(R) = false$.

A safety property P is a predicate that satisfies the two following conditions: any run for which P does not hold has a partial run for which P does not hold; and P does not hold in every extension of a partial run where P does not hold. A liveness property P, on the other hand, is one such that any partial run has an extension for which P holds.

It was shown in [17, 1] that any property can be expressed as the intersection of a safety and a liveness properties. Given a property P, possibly a set of properties (i.e., a problem), we denote by $S(P)$ the safety part of P and $L(P)$ the liveness part of P.

We capture in the following the safety aspect of indulgence through the notions of *stretched extension* and *unconscious* algorithms, which we introduce below. Let A and A' be any two algorithms.

- A' is a *stretched extension* of A if (a) A' is an extension of A and (b) $\forall R \in^\star A'$, $R \in^\star A$. (Every partial run of A' is a partial run of A.)

Notice that the notions of strict and stretched extensions are orthogonal. Algorithm A' might be a strict (resp. stretched) extension of A but not a stretched (resp. strict) extension of A.

Safety and unconsciousness

By the very definition of safety, we immediately get the following:

Proposition 2. *If the stretched extension A' of an algorithm A solves a problem P then A' ensures $S(P)$.*

Proof. (Sketch) Assume by contradiction that A' does not ensure $S(P)$. By definition of safety, there is a partial run $R \in^\star A'$, such that $S(P)$ does not hold in R, nor in any extension of R. Because A' is a stretched extension of A, $R \in^\star A$, which implies that A does not solve P.

This property is interesting because it helps expresses the fact that, if an indulgent algorithm A solves some problem P, while relying on some information about the interleaving of the processes, then A preserves the safety part of P even if the information turns out not to be accurate. The stretched extension of A precisely captures the situation where this information is not accurate. We say that the algorithm resulting from this situation is *unconscious*.

Definition (unconsciousness). Algorithm A is *unconscious* if every run R is such that $R \in A$ if every partial run R_i of R is such that $R_i \in^\star A$.

Indulgent algorithms like in [7, 19, 4, 23, 21, 8] are *conscious* because they rely on eventual information about at least one interleaving I. Any such algorithm A tolerates an interleaving I with a run $R \notin A$ such that $I(R) = I$ and all partial runs of R are in A. For instance, shared memory asynchronous algorithms are both indulgent and unconscious. Eventually synchronous algorithms are, on the other hand, indulgent but conscious. Indeed, consider a run R where every process p_i takes steps in rounds i, i^2, i^3, etc. Every partial run of R is eventually synchronous. However, R is not. Interestingly, by the definitions of the notions of stretched extensions and unconscious algorithm, we immediately get:

Proposition 3. *The stretched extension of any algorithm is an unconscious algorithm.*

For instance, the stretched extension of an eventually synchronous algorithm is asynchronous.

Proposition 2 and Proposition 3 say that if A solves some problem P while relying on some information about the interleaving of the processes (e.g., A assumes eventual synchrony), then A preserves the safety part of P even if the information turns out not to be accurate (e.g., even if the system ends up being asynchronous).

6 Uniformity

In the following, we show that indulgent algorithms are inherently *uniform*, in the intuitive sense that they are not sensitive to safety properties that restrict only the behavior of correct processes (which we call *correct-restrictions)*. We will illustrate the idea of uniformity through the consensus problem and point

out the fact that uniformity does not hold for algorithms that are not indulgent. Later, we will use the notion of uniformity to prove that certain problems do not have indulgent solutions. We first introduce below the notion of a *correct-restriction* of a property.

Correct restriction of a property

Informally, the *correct-restriction* of P, denoted $C(P)$, is the restriction of P to correct processes.

Definition (Correct-restriction). Let P be any property, we define the *correct-restriction* of P, denoted $C[P]$, as follows. For any run R, $C[P](R) = true$ if and only if $\exists R'$ such that $\forall p \in correct(R)$, $R/p = R'/p$ and $P(R') = true$.

Proposition 4. *Let P be any safety property and A any indulgent algorithm. If A satisfies $C[P]$ then A satisfies P.*

Proof. (Sketch) Let P be any safety property and A any indulgent algorithm that satisfies $C[P]$.

Assume by contradiction that A does not satisfy P. This implies that there is a run of A, say R, such that $P(R)$ is false. Because P is a safety property, there is a partial run of R, R', such that $P(R')$ is false.

By the indulgence of A, and our assumption that any interleaving has a failure-free extension, A has an extension of R', say R'', where all processes are correct.

Because P is a safety property and $P(R')$ is false, $P(R'')$ is also false. Hence, $C[P](R'')$ is false because all processes are correct in R'' and $C[P](R'') = P(R'')$. A contradiction with the fact that A satisfies $C[P]$.

Example: consensus

An immediate corollary of Proposition 4 concerns for instance the *consensus* [9] and *uniform consensus* problems (resp. *total order broadcast* and *uniform total order broadcast*) [14]. Before stating our corollary, we recall below the consensus problem.

We assume here a set of values V. For every value $v \in V$ and every process $p \in \Pi$, there is an initial state e_p of p associated with v and e_p is no associated with any other value $v' \neq v$; v is called the initial value of p (in state e_p). Hence, each vector of n values (not necessarily different ones) correspond to an initial configuration of the system. We also assume that, among other distributed services used by the processes, a specific one models the act of *deciding* on a value. The service, called the *output* service, has an operation *output()*; when a process p invokes that operation with an input parameter v, we say that p decides v.

An algorithm A solves the consensus problem if, in any run $R = (C, S)$, the three following properties are satisfied.

– *Validity:* the value decided by any process p_i in R is the initial value of some process p_j in C.
– *Agreement:* no two processes decide different values in R;
– *Termination:* every correct process in R eventually decides in R.

Clearly, agreement and validity are safety properties whereas termination is a liveness property. Two weaker, yet orthogonal, variants of *consensus* have been studied in the literature. One, called *non-uniform consensus*, only requires that no two *correct* processes decide different values. (May be counter intuitively, this is a liveness property.) Another variant, called *k-agreement* [5], requires that the number of different values decided by all processes (in any run) is at most k.

The following is a corollary of Proposition 4.

Corollary 1. *Any indulgent algorithm that solves consensus also solves uniform consensus.*

This is not the case with non-indulgent algorithms as we explain below. Consider a system of 2 processes $\{p_1, p_2\}$ using two services: an atomic shared register and a perfect failure detector. The latter service ensures that any process is eventually informed about the failure of the other process and only if the other process has indeed failed. The idea of a non-indulgent algorithm solving non-uniform consensus is the following: process p_1 decides its initial value and then writes it in the shared register; process p_2 keeps periodically consulting its failure detector and reading the register until either (a) p_1 is declared faulty by the failure detector or (b) p_2 reads p_1's value. In the first case (a) p_2 decides its own value and in the second (b) p_2 decides the value read in the register. If both processes are correct, they both decide the value of p_1. If p_1 fails after deciding, p_2 might decide a different value.

7 Failure sensitivity

In the following, we show that no indulgent algorithm can solve certain problems if at least one process can fail, even if this process can do so only initially, i.e., if the algorithm is $1^\star$-resilient. To simplify, we call a $1^\star$-resilient indulgent algorithm simply a $1^\star$-indulgent algorithm.

The problems we show impossible are those we call *failure sensitive*. In short, these are problems that resemble consensus with the particularity that the decision value might be considered valid depending on whether certain processes have failed. These problems include several classical problems in distributed computing like *terminating reliable broadcast*, *interactive consistency* and *non-blocking atomic commit* [14].

To prove our impossibility, we proceed as follows, we first define a simple failure sensitive problem, which we call *failure signal*, and which we show is impossible with a $1^\star$-indulgent algorithm. Then we show that any solution to *terminating reliable broadcast*, *interactive consistency* or *non-blocking atomic commit* solves *failure signal*: in this sense, *failure signal* is weaker than all those problems which are thus impossible with a $1^\star$-indulgent algorithm.

The failure signal problem

In failure signal, just like in consensus, the goal is for processes to decide on a value based on some initial value. As we explain however, unlike consensus, no agreement is required and a process can decide different values.

More specifically, in failure signal, a specific designated process p has an initial binary value, 0 or 1, as part of p's initial state. The two following properties need to be satisfied: (1) every correct process eventually decides and (2) no process (a) decides 1 if p proposes 0, nor (b) decides 0 if p proposes 1 and p is correct.

Interestingly, we prove the impossibility of *failure signal* by reduction to our *uniformity* result (Proposition 4). We prove by contradiction that, if there is a $1^\star$-indulgent algorithm that solves failure signal, then there is an algorithm that ensures the corrected-restriction of a safety property, without ensuring the actual property.

Proposition 5. *There is no solution to failure signal using a $1^\star$-indulgent algorithm.*

Proof. (Sketch) Assume by contradiction that there is a $1^\star$-indulgent algorithm that solves failure signal. Consider the designated process p and some other process q. (Remember that we assume a system of at least two processes).

Define property P such that $P(R)$ is *false* in every run R where p proposes 1 and q decides 0 and *true* in all other runs. By definition of a correct-restriction, $C[P]$ is *false* in runs where p proposes 1, q decides 0 and all processes are correct, and *true* in all other runs.

We now show that if there is a $1^\star$-indulgent algorithm that solves failure signal, then A ensures $C[P]$ but not P.

It is easy to show that A ensures $C[P]$. Indeed, because A solves *failure signal*, in any run R where p proposes 1 and all processes are correct, all processes decide 1.

We now show that A does not ensure P. Remember that A is a $1^\star$-resilient algorithm: A tolerates at least one initial failure. Consider a run R where p proposes 0 and does not take any step whereas all other processes are correct (p initially fails). Any $1^\star$-resilient algorithm that solves the failure signal problem has such a run R. In this run, every process that decides decides 0.

Consider now a run R' with the same schedule as R, except that p initially proposes 1 (and fails before taking any step). Such a run R is also a run of A and, because no process else that p, which fails initially, can distinguish R from R', all processes but p decide 0. This run R' is thus a run of A and $P(R')$ is false. This contradicts the uniformity of A.

Example 1: terminating reliable broadcast

In *terminating reliable broadcast*, also called *Byzantine generals*, a specific designated process is supposed to *broadcast* one message $m \neq \perp$ that is a priori unknown to the other processes. (In our model, the process invokes a specific service with m as a parameter.) In a run R where the sender p does not fail, all

correct processes are supposed to eventually receive m. If the sender fails, then the processes might or not receive m. If they do not, then they receive a specific message $\perp$ indicating that the sender has failed. More specifically, the following properties need to be satisfied. (1) Every correct process eventually receive one message; (2) No process receives more than one message; (3) No process receives a message different from $\perp$ or the message broadcast by the sender; (4) No two processes receive different messages; and (5) No process receives $\perp$ if the sender is correct.

The following is a corollary of Proposition 5.

Corollary 2. *No $1^\star$-resilient algorithm solves terminating reliable broadcast.*

Proof. (Sketch) We simply show how any solution to *terminating reliable broadcast* can be used to solve *failure signal*. Assume there is an algorithm A that solves terminating reliable broadcast. Whenever the designated process p (in failure signal) proposes a value, 0 or 1, p broadcasts a message with that value to all, using *terminating reliable broadcast*. Any process that receives the message delivers the value in the message (0 or 1). A process that delivers $\perp$ decides 0.

Example 2: non-blocking atomic commit

In non-blocking atomic commit, processes do all start with initial values 0 or 1, and are supposed to eventually decide one of these values. The following properties need to be satisfied. (1) Every correct process eventually decides one value (0 or 1); (2) no process decides two values; (3) No two processes decide different values; (4) No process decides 1 if some process proposes 0 and no process decides 0 if all processes propose 1 and no process fails.

The following is a corollary of Proposition 5.

Corollary 3. *No $1^\star$-resilient algorithm solves non-blocking atomic commit.*

Proof. (Sketch) Assume there is a solution to non-blocking atomic commit. We show how to obtain a solution to failure signal. All processes but p propose 1. Process p proposes exactly its initial value (of failure signal) to non-blocking atomic commit. The processes decide the output of non-blocking atomic commit. Because all processes but p propose 1, the decision can be 1 only if p proposes 1, and can be 0 only if p fails or proposes 0.

Example 3: interactive consistency

In interactive, processes do all start with initial values, and are supposed to eventually decide a n-vector of values. The following properties need to be satisfied. (1) Every correct process eventually decides one vector; (2) No process decides two vectors; (3) No two processes decide different vectors; (4) If a process

decides a vector v, then $v[i]$ should contain the initial value of p_i if p_i is correct. Otherwise, if p_i is faulty, $v[i]$ can be the initial value of p_i or $\perp$.

The following is a corollary of Proposition 5.

Corollary 4. *No $1^\star$-indulgent algorithm solves interactive consistency.*

Proof. (Sketch) Assume there is a solution to interactive consistency. Assume p is p_i. We show how to obtain a solution to failure signal. All processes propose to interactive consistency their identity, except p which proposes its initial value of failure signal. If a process q outputs a vector v such that $v[i] \neq \perp$, then q decides $v[i]$. Else, q decides 0.

8 Divergence

We now capture, in a general way, the traditional partitioning argument that is frequently used in distributed computing, e.g., [2]. This argument was traditionally used for message passing asynchronous algorithms where half of the processes can fail. In this case, the system can partition into two disjoint subsets that progress concurrently. We precisely state it here in the context of indulgent algorithms using timeless services which, as we pointed out, is a wider class than the class of asynchronous ones using message passing, and for systems with several possible partitions (the case with two partitions is just one particular case).

Definition (divergent property). We call a $k-$*divergent* property P a property such that for any k disjoint non-empty subsets of processes Π_1, Π_2,..Π_k, there is a configuration C such that every k runs R_1, R_2,..R_k of A, such that R_i involves only processes from Π_i, have respective partial runs R'_1, R'_2,..,R'_k for which $S(P(R'_1.R'_2...R'_k))$ is false.

Remember that $S(P)$ denotes the safety part of P. We call configuration C the *critical configuration* for Π_1, Π_2,..Π_k with respect to P. Note that, by construction, any property that is $k-$divergent is also $k+1-$divergent.

To intuitively illustrate the idea of a $2-$divergent property, consider the specification of consensus in a system of 2 processes p_1 and p_2. Consider the initial configuration where p_1 has initial value 1 and p_2 has initial value 2. Starting from C, every run R_1 involving only p_1 eventually decides 1 and every run R_2 involving only p_2 eventually decides 2. Consider the partial run R'_1 of R_1 composed of all steps of R_1 until the decision of p_1 (1) is made, and the partial run R'_2 of R_2 until the decision of p_2 (2) is made. Clearly, the safety of consensus (in particular *agreement*) is violated in $R'_1.R'_2$.

Definition (timeless service). We say that an algorithm A uses *timeless* services if for any two partial runs R_1 and R_2 of A starting from the same initial configurations C and involving disjoint subsets of processes, if A has an extension of R_1, $R_1.R'_1$ such that $I(R'_1) = I(R_2)$, then $R_1.R_2$ is also a run of A.

Examples of timeless services include sequentially consistent shared objects [18] as well as reliable message passing or broadcast primitives [14]. To illustrate the underlying idea, consider an algorithm A in a system of 2 processes p_1 and p_2 using a message passing primitive which ensures that any message sent from process p_1 to process p_2 is eventually received by p_2, provided p_2 is correct. Assume that A has a partial run R_1 where p_1 executes steps alone and a partial run R_2 where p_2 executes steps alone. (Clearly, p_2 cannot have received any message from p_1 in R_2.) Provided that A does not preclude the possibility of p_2 to execute steps alone after R_1, and because there is no guarantee on the time after which the message of p_1 arrives at p_2, then $R_1.R_2$, the composition of both partial runs, is also a possible run of A. This captures the intuition that the message of p_1 can be arbitrarily delayed.

Proposition 6. *No $(n-\lfloor n/x \rfloor)$-indulgent algorithm ensures a $x-$*divergent *property using $x-$*timeless *services.*

Proof. (Sketch) Assume by contradiction that there is a $(n - \lfloor n/x \rfloor)$-resilient indulgent algorithm A that ensures a $x-$*divergent* property P using *timeless* services.

Divide the set of processes Π of the system into k subsets Π_1, Π_2,..Π_x of size at least $\lfloor n/x \rfloor$ such that all the subsets are disjoint and their union is Π. Consider the critical configuration C for Π_1, Π_2,..Π_x with respect to P.

Because the algorithm A is $(n - \lfloor n/x \rfloor)$-resilient, and each Pi_i is of size at least $\lfloor n/x \rfloor$, then A has x runs R_1, R_2,..R_x such that each such R_i involves only processes in Π_i, i.e., only processes of P_i take steps in R_i and every such R_i start from C.

Because P is $x-$*divergent*, these runs have respective partial runs R'_1, R'_2,..,R'_k such that $S(P(R'_1.R'_2...R'_k))$ is false. We need to show that $R'_1.R'_2...R'_k$ is also a partial run of A. Because $S(P(R'_1.R'_2...R'_k))$ is false, this would contradict the very fact that A ensures P.

We first show that $R'_1.R'_2$ is a partial run of A. By the assumption that A is $(n-\lfloor n/x \rfloor)$-resilient, there is a partial run R_0 of A such that $I(R_0) = I(R'_1.R'_2)$. (Remember that a x-resilient algorithm is one that tolerates *all* interleavings where at least $n - x$ processes appear infinitely often).

By the indulgence of A, there is a partial run R''_2 such that $R'_1.R''_2$ is a partial run of A and $I(R'_1.R''_2) = I(R'_1.R'_2)$. By the assumption that A uses timeless services, $R'_1.R'_2$ is also a partial run of A. By a simple induction, $R'_1.R'_2...R'_k$ is also a run of A.

Because $S(P(R'_1.R'_2...R'_k))$ is false, P is false in every extension of $R'_1.R'_2...R'_k$: contradiction.

The following is a corollary of Proposition 6.

Corollary 5. *No $(n - \lfloor n/2 \rfloor)$-indulgent algorithm using message passing or sequentially consistent objects can implement a safe register.*

There are non-indulgent algorithms that implement a safe register with any number of failures and using only message passing. For instance, an algorithm

assuming a perfect failure detector. The idea is to make sure every value written is stored at all processes that are not detected to have crashed and the value read can then simply be a local value. On the other hand, Corollary 5 means that an algorithm using eventually perfect failure detectors, and possibly also sequentially consistent registers or message passing, cannot implement a safe register if two disjoint subsets of processes can fail. This clearly also applies to problems like consensus.

The following is also a corollary of Proposition 6.

Corollary 6. *No $(n - \lfloor n/k \rfloor)$-indulgent algorithm using message passing or sequentially consistent objects can solve k-set agreement [5].*

9 Concluding remarks

This paper presents a general characterization of indulgence. The characterization does not require any failure detector machinery [4], or timing assumptions [7]. It is furthermore not restricted to a specific communication scheme. Instead, we consider a general model of distributed computation where processes might be communicating using any kind of services, including shared objects, be they simple read-write registers [18], or more sophisticated objects like compare-and-swap or consensus [15], as well as message passing channels and broadcast primitives [14].

May be interestingly, our characterization of indulgence abstracts the essence of the notion of *unreliable failure detection*. This notion, informally introduced in [4], captures the idea that failure detectors do not need to be accurate to be useful in solving interesting problems. This notion has however never been precisely defined.[4] Using our characterization, we can precisely define it by simply stating that a failure detector is *unreliable* if any algorithm that uses that failure detector is indulgent.

Generalizing the notion of a failure detector, one could also consider oracles that inform a process that certain processes will be scheduled before others. (Say an oracle that declares a run as being eventually synchronous.) Our characterization of indulgence also helps captures what it means for such oracles to be unreliable.

To conclude, it is important to notice that we focused in this paper on the computability of indulgent algorithms and did not discuss their complexity. There are many interesting open problems in measuring the inherent overhead of indulgence. This goes first through defining appropriate frameworks to measure the complexity of indulgent algorithms, e.g., [6, 16, 25].

References

1. Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[4] Except in [10] in the specific message passing context.

2. Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message passing systems. *Journal of the ACM*, 42(2):124–142, January 1995.
3. Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.
4. Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.
5. Soma Chauduri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, 1993.
6. Partha Dutta and Rachid Guerraoui. The inherent price of indulgence. In *PODC '02: Proceedings of the annual ACM symposium on Principles of distributed computing*, pages 88–97, 2002.
7. Cynthia Dwork, Nancy A. Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.
8. Christof Fetzer, Ulrich Schmid, and Martin Susskraut. On the possibility of consensus in asynchronous systems with finite average response times. In *International Conference on Distributed Computing Systems*, pages 271–280. ieee, 2005.
9. Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.
10. Rachid Guerraoui. Indulgent algorithms. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, Portland, Oregon, USA*, pages 289–297. ACM, July 2000.
11. Rachid Guerraoui. On the hardness of failure sensitive agreement problems. *Information Processing Letters*, 79, 2001.
12. Rachid Guerraoui. Non-blocking atomic commit in asynchronous distributed systems with failure detectors. *Distributed Computing*, 15(1):17–25, 2002.
13. Rachid Guerraoui and Michel Raynal. The information structure of indulgent consensus. *IEEE Trans. Computers*, 53(4):453–466, 2004.
14. Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In Sape J. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. Addison-Wesley, 1993.
15. Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):123–149, January 1991.
16. Idit Keidar and Alex Shraer. Timeliness, failure detectors and consensus peformance. In *PODC '06: Proceedings of the annual ACM symposium on Principles of distributed computing*, New York, NY, USA, 2006. ACM Press.
17. Leslie Lamport. Proving the correctness of multiprocessor programs. *Transactions on software engineering*, 3(2):125–143, March 1977.
18. Leslie Lamport. How to make a multiprocessor computer that correct executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.
19. Leslie Lamport. The Part-Time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.
20. Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
21. Achour Moustefaoui, Michel Raynal, and Corentin Travers. Crash-resilient time-free eventual leadership. In *Proceedings of the International Symposium on Reliable Distributed Systems*, pages 208–217. IEEE, 2004.
22. Livia Sampaio and Francisco Brasileiro. Adaptive indulgent consensus. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 422–431, 2005.

23. Gadi Taubenfeld. Computing in the presence of timing failures. In *Proceedings of theInternational Conference on Distributed Computing Systems (DCS)*, 2007.
24. Pedro Vicente and Luis Rodrigues. An indulgent uniform total order broadcast algorithm with optimistic delivery. In *Proceedings of the International Symposium on Reliable Distributed Systems (SRDS)*, pages 92–80, 2002.
25. Piotr Zielinski. Optimistically terminating consensus. In *Proceedings of the Symposium on Parallel and Distributed Computing*, 2006.